

A simple implementation of set algorithms for the STL

Philip Bille

*Department of Computer Science, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen East, Denmark*
beetle@diku.dk

Abstract. We consider the implementation of algorithms for manipulating sets with the purpose of creating an implementation for the Copenhagen STL.

1. Introduction

The concept of a set and the basic operations on sets is a fundamental data structure in computer science and mathematics. It is therefore essential that we provide efficient representation of sets and implement fast algorithms to manipulate them. In the following sections we do exactly this and discuss the results of our work.

2. The problem

In our case we represent sets by *ordered* sequences and we wish to support the following operations:

- $A \subset B$. Decide whether or not all the elements of A are *included* in B .
- $A \cup B$. The *union* of A and B are the elements that are in A or B .
- $A \cap B$. The *intersection* of A and B are the elements that are in A and B .
- $A \setminus B$. The *difference* of A and B are the elements that are in A but not in B .
- $A \Delta B$. The *symmetric difference* of A and B are the elements that are in A and B but not both.

The input sequences are not allowed to be modified.

2.1 A simple solution

The operations described above are very similar and since the sequences are ordered it is easy to create linear time algorithms that iteratively constructs the output by maintaining pointers in the sequences and advancing them based on comparisons. This yields a simple general implementation as in figure 1.

```

template<class In1, class In2, class Out>
Out operation(In1 first1, In1 last1,
              In2 first2, In2 last2, Out result) {
    while (first1 != last1 && first2 != last2) {
        if (*first1 < *first2)
            // Modify result and advance some iterators
        else if (*first2 < *first1)
            // Modify result and advance some iterators
        else
            // Modify result and advance some iterators
    }
    return // appropriate result
}

```

Figure 1. A simple implementation

Where `operation` is one the above set operations. The `first` and `last` parameters are iterators marking the beginning and end of each of the input sequences. `result` mark the output. The `includes` operation differ slightly from this form since it only produces a boolean value.

If we advance at least one pointer in each iteration at most $2 \cdot (\text{last1} - \text{first1}) + \text{last2} - \text{last1} - 1$ comparisons are used. This is exactly the requirement imposed by the C++ standard.

It is clear that this is the intended implementation of the operations and it is also precisely the implementation provided by SGI STL. Since the input must not be modified the algorithms indeed have to use linear time to construct the output and we can only hope to improve some constant factors. Under less strict circumstances the problem of performing the union operation can actually be done in optimal $O(n \lg(m/n))$, where m and n are the size of the sets [1].

2.2 Minor improvements

The `while` loop is usually compiled to a conditional jump at the beginning of the loop and an unconditional jump backwards at the end of the loop. If we modify the `while` loop in figure 1 to a `do-while` loop as in figure 2 we can save the unconditional jump since a conditional jump at the end of the loop clearly suffices. Additionally the efficiency would naturally be improved by perform loop unrolling. Both of these optimizations can however be performed by most compilers and we have tested both without any evident improvement. We therefore leave the code as in figure 1 since this is clearly the most readable.

```
template<class In1, class In2, class Out>
Out operation(In1 first1, In1 last1, In2 first2,
              In2 last2, Out result) {
    if (first1 == last1 || first2 == last2) then return;
    do {
        // same as original
    } while (first1 != last1 && first2 != last2);
    return // appropriate result
}
```

Figure 2. do-while version.

3. Conclusion

We have constructed a simple solution to the problem. Our implementation is very similar to the SGI implementation and tests have revealed that the speed is approximately the same. We believe that is probably the best general solution but of course in certain conditions improvements might be possible.

References

- [1] R. E. T. Mark R. Brown, A fast merging algorithm, *Journal of the ACM* **26** (1979), 211-226.

4. Program

Below is listed the source of the components.

```
#include <iterator>
#include <algorithm>

template <typename initerator1, typename initerator2>
inline bool STL_includes(initerator1 first1, initerator1 last1,
                        initerator2 first2, initerator2 last2) {
    while (first1 != last1 && first2 != last2)
        if (*first2 < *first1)
            return false;
        else
            if(*first1 < *first2)
                ++first1;
            else
                ++first1, ++first2;
    return first2 == last2;
}
```

```

template <typename initerator1, typename initerator2, typename comp >
inline bool STL_includes(initerator1 first1, initerator1 last1,
    initerator2 first2, initerator2 last2, comp cmp) {
    while (first1 != last1 && first2 != last2)
        if (cmp(*first2, *first1))
            return false;
        else if(cmp(*first1, *first2))
            ++first1;
        else
            ++first1, ++first2;
    return first2 == last2;
}

```

```

template <typename initerator1, typename initerator2, typename outiterator>
outiterator STL_set_union(initerator1 first1, initerator1 last1,
    initerator2 first2, initerator2 last2,
    outiterator result) {
    while (first1 != last1 && first2 != last2) {
        if (*first1 < *first2) {
            *result = *first1;
            ++first1;
        }
        else if (*first2 < *first1) {
            *result = *first2;
            ++first2;
        }
        else {
            *result = *first1;
            ++first1;
            ++first2;
        }
        ++result;
    }
    return copy(first2, last2, copy(first1, last1, result));
}

```

```

template <typename initerator1, typename initerator2, typename outiterator,
    typename comp>
outiterator STL_set_union(initerator1 first1, initerator1 last1,
    initerator2 first2, initerator2 last2,
    outiterator result, comp cmp) {
    while (first1 != last1 && first2 != last2) {
        if (cmp(*first1, *first2)) {
            *result = *first1;
            ++first1;
        }
    }
}

```

```
else if (comp(*first2, *first1)) {
    *result = *first2;
    ++first2;
}
else {
    *result = *first1;
    ++first1;
    ++first2;
}
++result;
}
return copy(first2, last2, copy(first1, last1, result));
}

template <typename initerator1, typename initerator2, typename outiterator>
outiterator STL_set_intersection(initerator1 first1, initerator1 last1,
                                initerator2 first2, initerator2 last2,
                                outiterator result) {
    while (first1 != last1 && first2 != last2)
        if (*first1 < *first2)
            ++first1;
        else if (*first2 < *first1)
            ++first2;
        else {
            *result = *first1;
            ++first1;
            ++first2;
            ++result;
        }
    return result;
}

template <typename initerator1, typename initerator2, typename outiterator,
          typename comp>
outiterator STL_set_intersection(initerator1 first1, initerator1 last1,
                                initerator2 first2, initerator2 last2,
                                outiterator result, comp cmp) {
    while (first1 != last1 && first2 != last2)
        if (cmp(*first1, *first2))
            ++first1;
        else if (cmp(*first2, *first1))
            ++first2;
        else {
            *result = *first1;
            ++first1;
            ++first2;
        }
}
```

```

        ++result;
    }
    return result;
}

```

```

template <typename initerator1, typename initerator2, typename outiterator>
outiterator STL_set_difference(initerator1 first1, initerator1 last1,
                              initerator2 first2, initerator2 last2,
                              outiterator result) {
    while (first1 != last1 && first2 != last2)
        if (*first1 < *first2) {
            *result = *first1;
            ++first1;
            ++result;
        }
        else if (*first2 < *first1)
            ++first2;
        else {
            ++first1;
            ++first2;
        }
    return copy(first1, last1, result);
}

```

```

template <typename initerator1, typename initerator2, typename outiterator,
         typename comp>
outiterator STL_set_difference(initerator1 first1, initerator1 last1,
                              initerator2 first2, initerator2 last2,
                              outiterator result, comp cmp) {
    while (first1 != last1 && first2 != last2)
        if (cmp(*first1, *first2)) {
            *result = *first1;
            ++first1;
            ++result;
        }
        else if (cmp(*first2, *first1))
            ++first2;
        else {
            ++first1;
            ++first2;
        }
    return copy(first1, last1, result);
}

```

```

template <typename initerator1, typename initerator2, typename outiterator>
outiterator

```

```
STL_set_symmetric_difference(initerator1 first1, initerator1 last1,
                             initerator2 first2, initerator2 last2,
                             outiterator result) {
    while (first1 != last1 && first2 != last2)
        if (*first1 < *first2) {
            *result = *first1;
            ++first1;
            ++result;
        }
        else if (*first2 < *first1) {
            *result = *first2;
            ++first2;
            ++result;
        }
        else {
            ++first1;
            ++first2;
        }
    return copy(first2, last2, copy(first1, last1, result));
}
```

```
template <typename initerator1, typename initerator2, typename outiterator,
          typename comp>
outiterator
STL_set_symmetric_difference(initerator1 first1, initerator1 last1,
                             initerator2 first2, initerator2 last2,
                             outiterator result,
                             comp cmp) {
    while (first1 != last1 && first2 != last2)
        if (cmp(*first1, *first2)) {
            *result = *first1;
            ++first1;
            ++result;
        }
        else if (cmp(*first2, *first1)) {
            *result = *first2;
            ++first2;
            ++result;
        }
        else {
            ++first1;
            ++first2;
        }
    return copy(first2, last2, copy(first1, last1, result));
}
```